Generating Permutation Instructions from a High-Level Description

Manikandan Narayanan[1] and Katherine A. Yelick[1,2]

[1]Computer Science Division, University of California, Berkeley, CA 94720
[2]NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA 94720
{nmani,yelick}@cs.berkeley.edu

Abstract

Fine-grained data parallelism, from media extensions to full streaming or vector instruction sets, offer enormous performance potential, if they can be effectively used from the application level. One critical aspect of their design is the organization of the registers and the generality of operations that move data between registers. In this paper we focus on this data-movement problem and demonstrate that starting with a high-level description of a data-parallel application, we can automatically map certain data-movements in the program onto a regular set of vector permutation instructions. Our language and compiler are based on StreamIt from MIT, and our target machine is the VIRAM processor from Berkeley. We devise new intermediate representations and operators for analysing data-movements, and demonstrate our technique on two benchmarks. We show that data-movement operations give an enormous performance boost for the benchmarks, and the performance of our technique is close to, and sometimes better than, hand-coded assembly.

1 Introduction

As the computer industry heads towards an era with billion transistor chips, the desire to turn chip real estate into high performance will increasingly rely on chip-level parallelism. Yet evidence shows that there are diminishing benefits from increasing the amount of dynamically discovered instruction-level parallelism, so hardware designers are turning to explicitly parallel instruction sets, with the emphasis on data-parallelism. Commercial processors now include small-scale data parallelism in their SIMD media extensions, such as Intel's SSE [3] and PA-RISC's MAX-2 [9]. Research prototypes such as VIRAM [7], Raw [12], and Imagine [4] all take advantage of larger degrees of parallelism, both to increase arithmetic performance and to mask the effects of memory latency. In spite of the latency-hiding features, and in the case of VIRAM the tight integration with DRAM memory, temporal locality is still critical. The three research processors take different approaches to the data-movement problem, with Raw at one extreme providing arbitrary movement between "tiles," providing a kind of on-chip message passing machine model, and VIRAM at the other providing only a limited set of regular permutation patterns between vector registers in the register set.

In this paper, we explore the use of StreamIt language and compiler [2], which generates code for the Raw machine, and extend it to generate data-movement code in VIRAM, a quite different task than in Raw. We augment the StreamIt compiler with new intermediate representations and operators to collect and analyse data-movements between data-parallel 'parts' of an application. The revised compiler that does *data-movement analysis* is called StreamIt-to-VIRAM. The data-movement instructions in VIRAM provide only limited set of permutations to ease the hardware implementation [8]. But, we show that it gives some codes an enormous performance boost as it avoids using memory to rearrange data, and we show that automatic code generation is feasible using the StreamIt model. Previous work has used these instructions only from hand-coded assembly language [14, 1], and for the single special case of automatically generated code for reduction operations [7].

Using two benchmark problems, sorting and Fast Fourier Transforms, we show that the performance is competitive with the best hand-coding on VIRAM. The benchmarks were chosen so that they contain data rearrangements implementable using both memory accesses and data-movement instructions, and hence exhibit a tradeoff in the number of register operations per memory access. Our results have implications for compiler writers on commercial and research architectures, as the permutation instructures are quite similar to some SIMD media extensions such as Intel's SSE [3] and PA-RISC's MAX-2 [9]. The approach of describing applications in StreamIt as a high-level structure, called the stream-graph, greatly simplifies data-movement analysis when compared to a compiler for a language like C or Fortran. Furthermore, our results support the inclusion of this style of register permutations in instruction sets, both because automatic code generation is practical and effective, and this limited set of permutations is sufficient to see enormous performance gains.

We first introduce a motivating example (Section 2), and use it as a running example to explain the StreamIt-to-VIRAM compiler (Sections 3, 4, 5). We then present the results of this work (Section 6), and the contributions and some future directions for this work (Section 7).

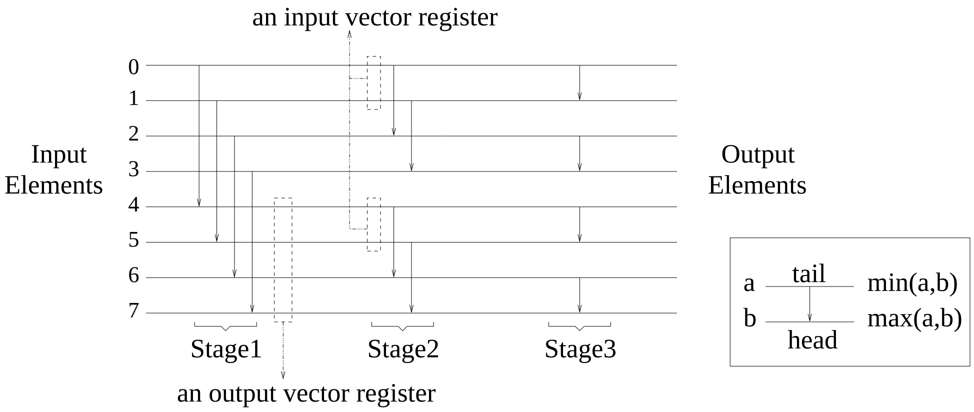

Figure 1: A comparator network CN for 8-element sequences. Each arrow represents a comparator, and the inset shows the function of a single comparator. Readers familiar with bitonic sequences would recognize this network as a bitonic merger that sorts an input bitonic sequence of length 8 [15].

2 Motivating Example and Background

This section introduces a simple example application, and uses it to provide a brief background on VIRAM and StreamIt. The application is a comparator network CN shown in Figure 1 [16]. Each stage of CN exhibits data-parallelism, and hence can be vectorized. Vectorizing one stage involves executing operations like "vector min" and "vector max" on the stage's two input vector registers to obtain the stage's two output vector registers. The stage's first input vector register holds the input element at the tail of the 4 comparators of the stage, while the stage's second input register holds the input element at the head of the comparators (some registers are marked in Figure 1; in this paper, a register simply means a vector register). The stage's two output registers similarly hold the output elements at the tail and head of the 4 comparators. The data rearrangement needed between one stage's output vector registers and the next stage's input vector registers

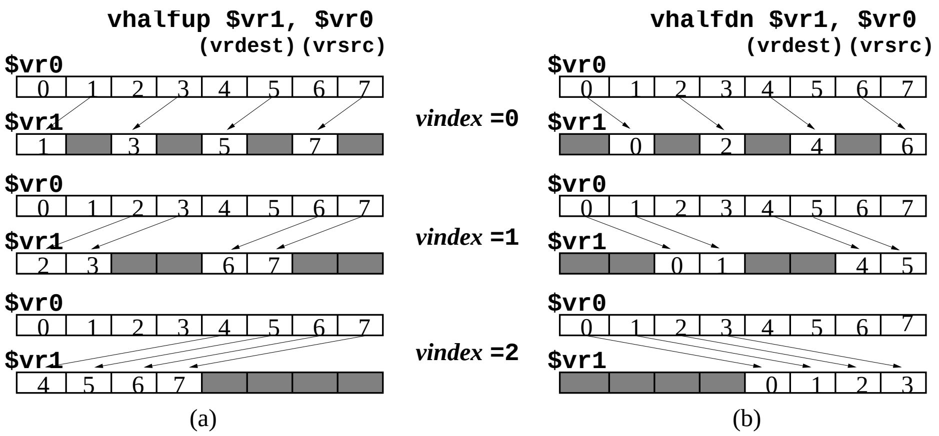

Figure 2: Permutations caused by vhalfup (a) and vhalfdn (b) instructions, illustrated with 8 elements per vector register. The figure is from [6]. Except for the direction of elements' motion, a vhalfup and vhalfdn instruction with the same programmable parameter (set in *vindex*) are similar. The instructions don't modify the shaded elements of the destination registers. The vector length must be a power of two for both the instructions.

are the subject of this work.

2.1 VIRAM

We describe vhalfup and vhalfdn instructions (let *vhalfdd* refer to either vhalfup or vhalfdn hereafter), to show that vhalfdd permutations on one stage's output registers can be used to setup the next stage's input register. Since vhalfdd permutations are more regular than arbitrary permutations, they require less chip area and execute fast [8]. We don't look at VIRAM's other data-movement instructions, as they are either general and hence execute slow (eg. vector compress and expand) [6], or are exploited by the existing compiler (eg. reduction operation). As shown in Figure 2, the vhalfdd instructions divides the source register into blocks of size $2 \cdot 2^{vindex}$, shifts half of the elements within each block by 2^{vindex} elements, and writes

```
void->void pipeline Main {
  add DataSource(8);
  add ComparatorStage(4, 1); //Stage1
  add MiddleStage;           //Stage2
  add ComparatorStage(4, 2); //Stage3
  add DataSink(8);
}
```

```
int->int filter Comparator {
  work push 2 pop 2 {
    int elem1 = pop();
    int elem2 = pop();
    int mink = min(elem1, elem2);
    int maxk = max(elem1, elem2);
    push(mink);
    push(maxk);
  }
}
```

```
int->int splitjoin ComparatorStage(int N, int W) {
  split roundrobin(W); //splitter
  for (int i=0; i<N; i++) {
    add Comparator;
  }
  join roundrobin(W);  //joiner
}
```

```
int->int splitjoin MiddleStage {
  split roundrobin(4); //splitter
    add ComparatorStage(2, 1); //Stage2a
    add ComparatorStage(2, 1); //Stage2b
  join roundrobin(4);  //joiner
}
```

Figure 3: StreamIt code for the comparator network CN from Figure 1. The *DataSource(n)* filter (code not shown) *only pushes* the n input elements from a "source_array", and the *DataSink(n)* filter (code not shown) *only pops* the n output elements into a "sink_array". A *roundrobin(W)* splitter, in one *round*, sends the first W items to the first stream in the splitjoin, the next W items to the second stream, and so on. A *roundrobin(W)* joiner has a similar function.

them into the corresponding position in the destination register.

2.2 StreamIt

We describe a subset of StreamIt here, and continue the running example from Figure 1. An application is expressed in StreamIt as a network of computational entities called filters. Filters communicate with neighboring filters via FIFO queues using the operations of $push(value)$ and $pop()$. The $work$ function that does the computation is invoked at every fine-grained execution of the filter, and the number of items pushed (popped) during each invocation, is the $push\ value$ ($pop\ value$). Figure 3 shows the code of our running example, and Figure 4 the corresponding stream-graph.

The high-level structures in this stream-graph are pipelines, which se-

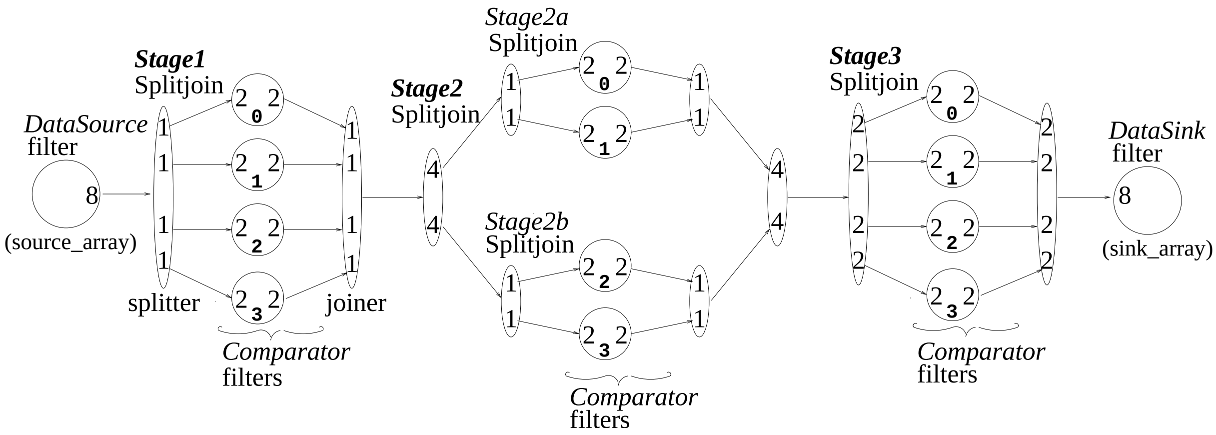

Figure 4: A stream-graph G_{CN} modelling the comparator network CN from Figure 1. The edges in the figure represent FIFO queues. The numbers inside a splitter or joiner are the roundrobin weights, and those near the output and input of each filter are the *push* and *pop* values respectively (this convention is followed in all figures). The small number near the bottom of each *Comparator* filter (a filter of *identity* 'Comparator') is the filter's 'position' in either of splitjoins, *Stage1*, *Stage2* or *Stage3*.

quentially compose streams (filters or other constructs), and splitjoins, which compose independent streams that diverge from a common *splitter* and merge into a common *joiner*. The splitter and joiner of a splitjoin are key places in the stream-graph where data-movement happens. StreamIt is a static language, i.e., the stream-graph structure (including the *push* and *pop* values, and roundrobin weights associated with roundrobin splitters and joiners) must be known at compile-time. A steady-state schedule is an ordering of the executions of filters, splitters, and joiners of the stream-graph that satisfies certain data-dependence semantics. A hierarchical steady-state schedule [2] is used as the *underlying schedule* for data-movement analysis in this work. The number of executions in the underlying schedule of a filter, splitter or joiner, is referred to as *multiplicity* in this paper.

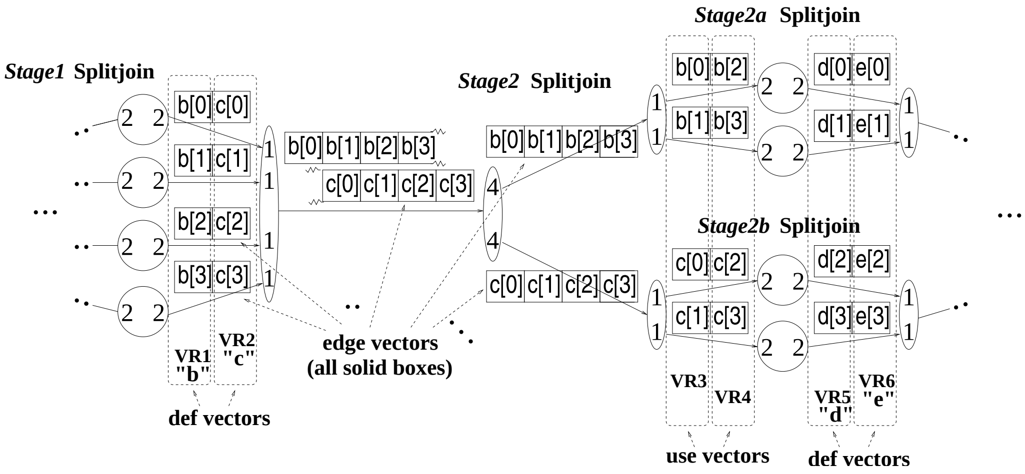

Figure 5: A section of the stream-graph of Figure 4 annotated with the underlying schedule's execution. The edge vectors represent the corresponding FIFO queues (the first element of the edge vector is the head of the FIFO queue, and so on). A *def vector* collects appropriate elements from edge vectors at the output of a vectorizable set, so it corresponds to an output vector register of the vectorizable set (this observation, which is similarly true for *use vectors*, follows from visualizing a data-parallel execution of the vectorizable set; see also Section 2).

3 Data-movement analysis

Data-movement analysis is the process of obtaining and analysing data-movements. First, we need a framework (a basis) for doing it, and this section presents such a framework (Section 3.2) inspired by a subset of features of StreamIt (Section 3.1). Our StreamIt-to-VIRAM compiler augments StreamIt with new representations and operators that help in data-movement analysis (they are described in this section and the next two).

3.1 Data-movement-specific features of StreamIt

The features of StreamIt used in our StreamIt-to-VIRAM compiler for data-movement analysis are listed below. These features could prove useful to extend data-movement analysis to other source languages like C.

1. Static nature: StreamIt's various static requirements let the stream-graph structure and steady-state schedule [2] to be determinable at compile-time. The concrete foundation provided by the stream-graph structure and (underlying) schedule greatly simplifies data-movement analysis.

2. Data-parallelism construct: A data-parallelism construct like splitjoin is important, since we are interested mainly in data-movements between data-parallel components of the application. A *vectorizable set* captures the notion of a data-parallel component, and is defined as a set of filters of same identity (hence same *work* function, *push* and *pop* values, etc.,) and same multiplicity that additionally satisfy the condition: these filters are the only independent filters in a splitjoin (called the enclosing splitjoin). The stream-graph in Figure 4 has three vectorizable sets that comprise the filters in $Stage1$, $Stage2$ and $Stage3$ splitjoins respectively (the enclosing splitjoin can also contain other splitjoins; eg. $Stage2$ splitjoin)

3. Data-centric abstractions: The *push* and *pop* FIFO queues and splitter and joiner abstractions in StreamIt are very much data-centric. Let's see their effect on data-movement analysis. We assume in this paper that the multiplicity of the filters of a vectorizable set is 1 (without loss of generality [11]). Then, each of these filters produces *push* values and consumes *pop* values in the underlying schedule. Hence, the vectorizable set as a whole has *push def vectors* and *pop use vectors*, with each def or use vector's length being the number of filters in the vectorizable set. Figure 5 illustrates different vectors, and also shows how def and use vectors correspond to output and input vector registers respectively of a vectorizable set. Thus, data-movement analysis now reduces to the problem of determining how to get a use vector from the def vector(s) that contributes data to the use vector.

4. Decomposable data-movements: A "global" data-movement captures how data gets routed between def vectors that contribute data to a use vector and the use vector itself (for example, the def vectors "*b*" and "*c*" contribute to the use vector *VR3* in Figure 5). A global data-movement is decomposable into a set of simple local data-movements at splitters or joiners that lie in the path in the stream-graph from the contributing def vectors to the use vector.

3.2 StreamIt-to-VIRAM's Data-movement framework

This section presents a new representation called *views* as part of a framework for doing data-movement analysis. *Vectorizable sets*, their *use* and *def vectors*, *edge vectors* (Figure 5), and *views* all constitute a natural framework or basis for doing data-movement analysis. If the user exposes communication patterns in the application over a suitably fine-grained stream-graph, then the data-movements in the stream-graph are readily available for analysis under this framework. We require the programmer to use only a subset of StreamIt [11] similar to the one described in Section 2.2. We believe that this subset has a good chance of covering parts of the application *that can benefit from vhalfdd permutations*. For example, the subset doesn't contain duplicate splitters, and vhalfdd permutations don't involve duplications.

A view of a edge or use vector (called as edge or use view) describes the composition of the vector by recording the "origin" (specific def vector) and "position or index in origin" of each element of the vector separately. An example of a edge view in Figure 5 is the ordered set $\{b[0], b[1], b[2], b[3], c[0], c[1], c[2], c[3]\}$ of the contents inside one edge vector. We exploit the contiguous regularity in this view to obtain a compact representation $\{(<b> + 0,\ +1,\ 4),\ (<c> + 0,\ +1,\ 4)\}$. Each structure of the form $(<x> + \mathit{offset},\ \mathit{stride},\ \mathit{length})$ in the view (we call it a subview), indicates a strided access on the vector x, and picks the $(\mathit{offset} + 0 \cdot \mathit{stride})$-th, $(\mathit{offset} + 1 \cdot \mathit{stride})$-th, $\ldots$, $(\mathit{offset} + (\mathit{length}-1) \cdot \mathit{stride})$-th elements of x. Subviews can also be "struct-strided" accesses [11]. Use views provide sufficient information for generating permutation instructions, and edge views provide enough information for obtaining the use views. Hence, views serve as an important intermediate representation and key component of data-movement framework.

4 Obtaining views

This section describes our extensions to the StreamIt compiler for obtaining the use views of all use vectors in the stream-graph, and forms the first phase of data-movement analysis. The task of obtaining use views is broken down into local tasks of obtaining edge views ("decomposable data-movements" feature in Section 3.1). The basic principle behind this decomposition strategy is the simulation of filter invocations and data-movements in the stream-graph, according to the underlying schedule. The simulation needs only the static values of the stream-graph, and the *work* functions of filters are abstracted out of the simulation. Though the simulation here is analogous to the one done by the StreamIt-to-Raw compiler [2], its purpose and implementation are quite different. The simulation here is for recognizing commu-

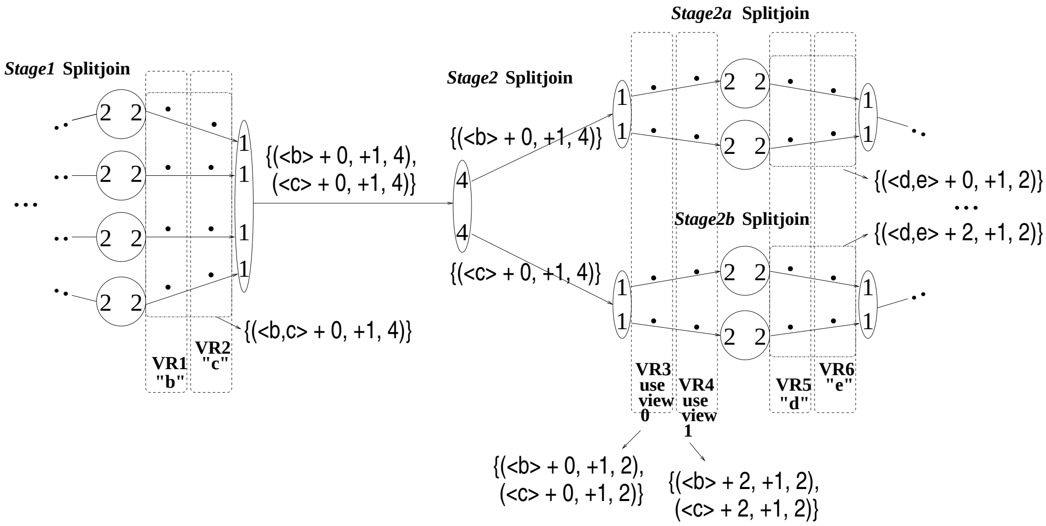

Figure 6: Obtaining edge and use views on a section of the stream-graph of Figure 4. Compare the views in this figure with the vectors in Figure 5.

nication patterns mappable to VIRAM's permutation instructions, but the one in StreamIt-to-Raw is for communication scheduling. The simulation in this work records the regularity structure in communication patterns using views and its associated operator *apply_stride*.

The apply_stride operator plays a key role in obtaining edge and use views. It applies a strided access on an input view to return an output view. In more detail, it outputs a view of a vector that is obtained by applying a strided access (specified by *offset*, *stride* and *length*) on the input vector (the use or edge vector described by the input view). For example, an apply_stride of $(\mathit{offset}, \mathit{stride}, \mathit{length})$ on an input view $\{(<a> + \mathit{offset}', \mathit{stride}', \mathit{length}')\}$ returns the output view, $\{(<a> + \mathit{offset} \cdot \mathit{stride}' + \mathit{offset}', \mathit{stride} \cdot \mathit{stride}', \mathit{length})\}$. For an elaborate description of apply_stride, see [11].

The edge views are obtained by traversing the edges of the stream-graph in a topological order (i.e., an edge is visited only after visiting all neighbouring edges coming into that edge). The current edge being visited could lie at the output of a splitter, joiner or filter, so the view of edge vector at the current edge is obtained by merely simulating the working of a splitter, joiner or filter, according to the underlying schedule. These simulations can be done efficiently using the apply_stride operator [11].

Once we have the edge views at a vectorizable set's input, we can readily obtain its use views. Specifically, the pth use view of the vectorizable set is an apply_stride of (p, $+pop$, $vlength$) on a view that is the concatenation of edge views at the vectorizable set's input (here *pop* is the *pop* value of a filter of this set, and $vlength$ is the number of filters in this set). Figure 6 illustrates the whole exercise of obtaining edge and use views, and is easily understood by comparing it with Figure 5.

5 Analysing views

Analysing views forms the second phase of data-movement analysis. Equipped with the use views of all use vectors returned by the first phase, we generate permutation instructions by looking for access patterns (or structure) in these views *that are mappable to vhalfdd permutations.* Instead of working directly with views, we work with an alternate characterization of a use vector's composition, called the *usage* and *location patterns* that are derived from use views. These patterns simplify the generation of vhalfdd (especially multiple vhalfdd) permutations, so we look at them first.

Consider a specific use vector UV. The usage and location pattern is defined for each def vector that contributes data to UV. Let contributed elements of a def vector refer to its data elements that end up in UV. Then, the position or index of each contributed element of a def vector *in the def vector itself* is the usage pattern of the def vector, and that *in the use vector* is the location pattern of the def vector. For example, if UV's composition is $\{a[0]$, $b[0]$, $a[1]$, $b[1]\}$, both a and b contribute their first two elements to UV and hence have the same usage pattern, which is the ordered set {0,1}. However, a and b's location pattern is the ordered set {0,2} and {1,3} respectively. To clarify, if the dth element of the def vector ends up in uth position of UV, then d and u are present (say, as the ith value) in the ordered set representing a def vector's usage and location pattern respectively. Having defined usage and location patterns using ordered sets, the next step is to compact these sets using a "block-stride" representation. A block-stride represents a usage or location pattern using four parameters (*offset*, *stride*, *length*, *blocking-factor*), as follows. Consider a vector of all possible positions or indices $\{0, 1, 2, \ldots\}$, and group every *blocking-factor* adjacent elements of this vector into blocks. The result is a new vector V of blocks, and a strided access of ($<V>$ + *offset*, *stride*, *length*) on V returns some blocks. The ordered set of positions in the returned blocks gives the pattern represented by this block-stride. For example, if an usage or location pattern is the ordered set {0,1,2,6,7,8,12,13,14}, then the pattern is represented by the block-stride (0, +2, 3, *blockf* 3) (*blockf* indicates *blocking-factor*). In this example, the vector V of blocks is $\{\underline{(0,1,2)}$, (3,4,5), $\underline{(6,7,8)}$, (9,10,11), $\underline{(12,13,14)}\}$, and the underlined blocks are spaced at a stride of

+2.

We now have the machinery to deduce or generate vhalfdd permutations from usage and location patterns that are represented as block-strides. A simple claim that lets us generate a single vhalfdd permutation follows. If **(A)** exactly two def vectors (allocated in registers *VRS1* and VRS2) contribute data to a use vector (allocated in *VRD*), **(B)** the usage pattern of both the def vectors w.r.t the use vector are (*offset*, +2, *length*, *blockf* 2^{vidx}) with *offset* being 0 or 1 and *length* being a power of two, and **(C)** the location pattern of the first and second def vector w.r.t the use vector are (0, +2, *length*, *blockf* 2^{vidx}) and (1, +2, *length*, *blockf* 2^{vidx}) respectively, **then** we can generate vhalfup(*VRD*, *VRS1*, *VRS2*, *vidx*) if *offset* is 0 or vhalfdn(*VRD*, *VRS2*, *VRS1*, *vidx*) if offset is 1, to setup data in the use vector. In the claim, vhalfdd(*vrdest*, *vrsrc1*, *vrsrc2*, *vindex*) is a shorthand for the two-instruction sequence: copy *vrdest, vrsrc1* and vhalfdd *vrdest, vrsrc2*. The claim is easy to verify, and applying it to Figure 6 shows that *VR3* can be setup with proper data by a vhalfup(*VR3*, *VR1*, *VR2*, 1), and *VR4* can be setup by a vhalfdn(*VR4*, *VR2*, *VR1*, 1).

Multiple vhalfdd permutations combine the effect of a list (i.e., sequence) of single vhalfdd permutations to achieve a complex permutation of data. Here is an example (VRS1 and VRS2 are source, VRD is destination, and VRT are temporary registers):

Pair 1: vhalfup(*VRT1*, *VRS1*, *VRS2*, 2), vhalfdn(*VRT2*, *VRS2*, *VRS1*, 2)
Pair 2: vhalfup(*VRT3*, *VRT1*, *VRT2*, 1), vhalfdn(*VRT4*, *VRT2*, *VRT1*, 1)
Last: vhalfup(*VRD*, *VRT3*, *VRT4*, 0)

A list of arbitrary single vhalfdd permutations is difficult to deduce, so we only consider a "constrained list" [11] of vhalfdd permutations (motivated by the Bitonic Sort application). A constrained list has three parameters, *orientation*, $vidx_{beg}$ and $vidx_{end}$. The above example is a constrained list with orientation 0, $vidx_{beg}$ 2, and $vidx_{end}$ 1 (orientation would be 1 if the "Last:" instruction in the example were a vhalfdn(*VRD*, *VRT4*, *VRT3*, 0) instead). A claim that lets us generate a constrained list, and is similar to the claim seen above, follows. If **(A)** exactly two def vectors contribute data to a use vector, **(B)** the usage pattern of both the def vectors are (*offset*, +2, *length*, *blockf* $2^{vidx_{end}}$) with *offset* being equal to *orientation* (and hence only 0 or 1) and *length* being a power of two, and **(C)** the location pattern of the first and second def vector are (0, +2, *length'*, *blockf* $2^{vidx_{beg}}$) and (1, +2, *length'*, *blockf* $2^{vidx_{beg}}$) respectively, **then** we can generate a constrained list of vhalfdd permutations with parameters, *orientation*, $vidx_{beg}$ and $vidx_{end}$ to setup data in the use vector. In the claim, $length \cdot 2^{vidx_{end}} = length' \cdot 2^{vidx_{beg}}$. By the claim, we can generate the example constrained list above if, *VRS1* and *VRS2* have the same usage pattern (0, +2, *length*, *blockf* 1), and location patterns (0, +2, *length'*, *blockf* 4) and (1, +2, *length'*, *blockf* 4) respectively, w.r.t *VRD*. The claim can be inductively proved by stepping through the constrained list's permutations and noticing their effect on the patterns.

6 Results

The StreamIt-to-VIRAM compiler described in this work is implemented by integrating the StreamIt compiler, which we modified to do data-movement analysis, and the existing VIRAM vectorizing C compiler *vcc*. The imple-

Method of Programming	Exploits vhalfdd permutations?	Performance (h/w peak: 1,600 MFlops)
C code and *vcc* compiler	No	94 MFlops
StreamIt-to-VIRAM compiler	Yes	1,109 MFlops
Hand-optimized assembly code based on [13]	Yes	898 MFlops

Table 1: Performance comparison of different methods of coding FFT in VIRAM on $N = 256$ *complex* elements (the *real* and *imaginary* parts are 32-bit floating point numbers).

mentation is a proof-of-concept, and the compilation process is fully automated except for two minor phases. Note that *no* assembly-level hand optimizations is done in these two phases [11], and there is no fundamental obstacle against automating these phases in future. The benchmark applications FFT [14, 17], and Bitonic Sort [15, 16] were considered to test our compiler. The benchmarks were chosen so that they contain data rearrangements implementable using both memory accesses and permutation instructions, and hence exhibit a tradeoff in the number of register-register operations per memory access.

6.1 Performance on VIRAM

VIRAM-1, a prototype VIRAM processor targeted to run at 200 MHz [7], was recently fabricated. It is not yet running in a test environment, and we obtain these results using a near cycle-accurate simulator. The performance of FFT on VIRAM is shown in Table 1, and two observations are immediate: Vhalfdd permutations are important in accelerating FFT (this agrees with the result in [14] on the superiority of vhalfdd permutations over other assembly optimizations for FFT), and our compiler beats even

Method of Programming	Exploits vhalfdd permutations?	Performance (h/w peak: 1,600 MOps)
C code and *vcc* compiler	No	19 MOps
StreamIt-to-VIRAM compiler	Yes	180 MOps
StreamIt-to-VIRAM compiler + cleanup	Yes	737 MOps
Hand-optimized assembly code used in DIS histogram benchmark [1]	Yes	738 MOps*

Table 2: Performance comparison of different methods of coding Bitonic Sort in VIRAM on $N = 128$ 32-bit integers. (* - This code sorts only 64 elements, and is part of the DIS histogram benchmark [1].)

the hand-optimized code of [13]. Our StreamIt-to-VIRAM compiler performs better than the hand-optimized code, because the former uses faster unit-stride vector loads [1] instead of indexed vector loads during the final few FFT stages. But, in order to use unit-stride loads, the FFT code produced by our compiler uses roughly 6 times more memory space than the hand-optimized code.

The results of Bitonic Sort are in Table 2. Hand-coding bitonic sort to exploit vhalfdd permutations is very tedious as a simple data-movement in the application can translate to multiple vhalfdd permutations in the assembly code (Section 5), and our StreamIt-to-VIRAM compiler provides a far better alternative. Some basic backend optimizations are absent in *vcc* [8], but a positive note in [8] says that these optimizations are "straightforward to add in future and have been available for years in all commercial compilers". Recall that the StreamIt-to-VIRAM compiler also uses *vcc*, and we tackle the absence of the basic backend optimizations via a "cleanup" (see Table 2) done by hand, to mainly remove spill and unwanted code from the final assembly file produced by the StreamIt-to-VIRAM compiler.

Compiler	FFT		Bitonic Sort	
	MFlops	Cycles	MOps	Cycles
StreamIt-to-Raw	90	28339 cycles	392	3425 cycles
StreamIt-to-VIRAM	1109	1846 cycles	180 / 737*	5968 / 1459* cycles

Table 3: Performance comparison of StreamIt-to-Raw and StreamIt-to-VIRAM compilers. For MFlops/MOPs calculation, the clock frequency used for Raw is 250 MHz, and VIRAM is 200 MHz. A recent paper [5] says that Raw core runs at 420 MHz at room temperature and nominal voltage. So, cycle counts are also given. The VIRAM numbers are same as before. (* - This number is StreamIt-to-VIRAM compiler + cleanup as in Table 2.)

6.2 Comparison with StreamIt-to-Raw

The StreamIt compiler [2] generates code for Raw, a communication-exposed architecture, and we extended it to generate data-movement code for VIRAM, a vector architecture. The two compilers, StreamIt-to-Raw and StreamIt-to-VIRAM, hence have very different responsibilities, but interestingly perform the analyses and transformations on the same IR, the stream-graph. One difference in compiler responsibility is generating vs scheduling communication operations (see also Section 4). The thrust of our StreamIt-to-VIRAM compiler is in *generating* permutation instructions, since each permutation instruction triggers a *hardcoded schedule* of basic data transfers on VIRAM's on-chip communication network.

Table 3 presents comparison between the two compilers on the same benchmarks (performance simulators of Raw and VIRAM were used to obtain the results). Since both compilers have the same frontend, we are infact comparing their backends. The StreamIt-to-VIRAM performs better on FFT. Some reasons for the current version of StreamIt-to-Raw backend's poor performance on FFT are imperfect load-balancing on mapping the stream-graph to Raw's tiles, and inter-tile synchronization overhead. On bitonic sort, the StreamIt-to-Raw performs better than StreamIt-to-VIRAM's 180 MOps result. But, as discussed before, if *vcc* is improved to include some basic backend optimizations, StreamIt-to-VIRAM should be better with 737 MOps. However, a caveat to note here is that Bitonic Sort is a suitable sorting algorithm for VIRAM, but not necessarily suitable for Raw.

7 Contributions and future work

This paper presented a *modular* approach for automatic code-generation of vhalfdd permutations in VIRAM. We showed that our approach performs competitively with hand-coded assembly and vhalfdd permutations provide an enormous performance boost for the benchmarks considered. Our modular approach demonstrates StreamIt's usefulness in data-movement analysis. Internal representations like views help achieve modularity by serving as clean interfaces between different phases of the compiler, which would be useful in retargeting to other vector or SIMD instruction sets.

A natural question at this juncture is: How *extensible* is the approach taken in this work to generate other permutation instructions? The answer is closely linked with the data-movement specific features of StreamIt (Section 3.1). Our approach could be extended to generate a permutation instruction, if the instruction's data rearrangement can be represented using the *structure* of the stream-graph (i.e., using the splitters and joiners that lie along the path from the relevant def vectors to the use vector). Hence, instructions with dynamic access patterns (eg. the powerful "vperm" instruction from Motorola's Altivec [10]) cannot be represented in this framework. But, interesting examples of instructions that could be represented are the "high or low unpack and interleave" instructions from Intel's SSE [3], and "mixR or mixL" instructions from PA-RISC's MAX-2 [9] (surprisingly, mixR and mixL are analogous to vhalfup and vhalfdn instructions respectively). All of these instructions add explicit parallelism to the instruction set, and history has shown that the ability to compile for such extensions is critical. Our work demonstrates that VIRAM-style regular permutation instructions are not only efficient from a hardware perspective, but can also be reasonably supported by compilers from a high-level parallel language like StreamIt.

Acknowledgements

We would like to thank members of the StreamIt group at MIT for letting us modify their compiler infrastructure. In particular, we would like to thank Bill Thies of the group for his constant support and co-operation, and instantaneous email replies! The Raw results in the paper were provided by him.

References

[1] B. Gaeke, P. Husbands, X. Li, L. Oliker, K. Yelick, and R. Biswas. Memory-Intensive Benchmarks: IRAM vs. Cache-Based Machines. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, Ft. Lauderdale, Florida, Apr 2002.

[2] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. Meli, A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, San Jose, California, Oct 2002.

[3] Intel Corporation. *The IA-32 Intel Architecture Software Developer's Manual (245470)*, 2003.

[4] B. Khailany, W. Dally, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media Processing With Streams. *IEEE Micro*, 21(2):35–46, Mar 2001.

[5] J. Kim, M. Taylor, J. Miller, and D. Wentzlaff. Energy Characterization of a Tiled Architecture Processor with On-Chip Networks. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 424–427, Seoul, Korea, Aug 2003.

[6] C. Kozyrakis. Scalable Vector Media-processors for Embedded Systems. Technical Report CSD-02-1183, Computer Science Division, University of California at Berkeley, May 2002.

[7] C. Kozyrakis, D. Judd, J. Gebis, S. Williams, D. Patterson, and K. Yelick. Hardware/compiler codevelopment for an embedded media processor. *Proceedings of the IEEE*, 89(11):1694–1709, Nov 2001.

[8] C. Kozyrakis and D. Patterson. Vector Vs Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 283–293, Instabul, Turkey, Nov 2002.

[9] R. Lee. Subword-parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, Aug 1996.

[10] Motorola, Inc. *Altivec Technology Programming Environments Manual*, 2002.

[11] M. Narayanan. Compiling Communication Access Patterns for a Vector Processor. Master's thesis, Computer Science Division, University of California at Berkeley, to appear, 2003.

[12] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–35, Mar 2002.

[13] R. Thomas. An Architectural Performance Study of the Fast Fourier Transform on Vector Iram. Technical Report CSD-00-1106, Computer Science Division, University of California at Berkeley, Aug 2000.

[14] R. Thomas and K. Yelick. Efficient FFTs On VIRAM. In *Proceedings of the 1st Workshop on Media Processors and DSPs*, Haifa, Israel, Nov 1999.

[15] http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm. Website Reference for Bitonic Sort.

[16] http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/sortieren.htm. Website Reference for Comparator Networks.

[17] http://www.cs.berkeley.edu/~demmel/cs267/lecture24/lecture24.html. Lecture Notes of the Course on Applications of Parallel Computers (CS267), Computer Science Division, University of California at Berkeley, Mar 1996.